

## **Defining Instruction Extensions in a Standard Programming Language**

### **CROSS-REFERENCE TO RELATED APPLICATIONS**

[01] The subject matter of this application is related to U.S. Application Serial Number 10/404,706 filed on March 31, 2003 and titled "Extension Adapter" which is incorporated herein by reference.

### **BACKGROUND OF THE INVENTION**

#### **Field of the Invention**

[02] The present invention relates generally to the field of programmable computer processors, and more particularly to application specific instruction sets.

#### **Description of the Prior Art**

[03] Computer processors can generally be sorted into two classes: general purpose processors that can be adapted to a multitude of applications; and application-specific processors that are optimized to serve specific applications. General purpose processors are designed to run a general instruction set, namely a set of instructions that the processor will recognize and execute. Such general instruction sets tend to include a large number of instructions in order to support a wide variety of programs. Application-specific processors are designed to run a more limited instruction set, where the instructions are more tailored or specific to the particular application. While an application-specific processor can enable certain programs to execute much faster than when run on a general purpose processor, they are by definition more limited in functionality due

to the limited instruction sets they run. Further, instructions for an application-specific processor must be defined before the processor is manufactured.

**[04]** Accordingly, what is desired is the ability to write a program in a convenient programming language and to extend an instruction set of a computer processor with instructions tailored to that program so that the program can execute on that computer processor more efficiently.

## **Defining Instruction Extensions in a Standard Programming Language**

### **BRIEF SUMMARY OF THE INVENTION**

[05] As general-purpose processors typically do not have programmable instruction sets, the present invention provides a method for programming a processor instruction set to include new instructions, and for replacing a critical code segment of a computer program with a function that causes the new instructions to execute. A programmable logic device (PLD) includes logic for enabling application-specific instructions (“instruction extensions”) to be stored and executed, so that a user can add new instructions that change with software on different implementations of the same silicon. These instructions are not hard-wired into processor core, but rather implemented using the programmably configurable logic of the PLD.

The present invention provides in various embodiments a system and method for revising a program to allow the program to execute on a processor system that includes a programmable logic device. In a method according to an embodiment of the present invention, a program is compiled to produce an executable file and an instruction is programmed into a programmable logic device of the processor system. The method includes profiling a program to identify one or more critical code segments, rewriting a critical code segment as a function, designating the function as code to be compiled by an extension compiler, replacing the critical code segment with a statement that calls the function, and compiling the revised program.

In one embodiment, compiling the program includes compiling the code with an extension compiler to produce a header file and an intermediate file that provides instructions for the programmable logic device. In another embodiment, compiling the program includes using a standard compiler to compile the remainder of the program together with a header file to generate an executable file.

Further aspects of the inventive method include evaluating the performance of the revised program, and comparing the performance to timing requirements or to prior performance. In one embodiment of the method, the function replacing the critical code segment is selected from a library of pre-defined functions. In another embodiment, the program is written in a program file and the function includes writing the code to an extensions file.

In a further embodiment, the program is written in a program file and designating the function as code to be compiled by an extension compiler includes writing the code into the program file and demarking the code. In a still further embodiment, compiling the revised program includes compiling an extensions file including the code to produce a header file and an intermediate file written in a hardware description language, for example in Verilog HDL.

A further understanding of the nature and advantages of the inventions herein may be realized by reference to the remaining portions of the specification and the attached drawings.

## **BRIEF DESCRIPTION OF DRAWINGS**

[06] FIG. 1 is a schematic diagram of an exemplary extensible processor system of the present invention;

[07] FIG. 2 is a schematic diagram of a programmable logic device (PLD) in accordance with the schematic of FIG. 1;

[08] FIG. 3 illustrates an example of the cluster block implementation illustrated in FIG. 2;

[09] FIG. 4 is a schematic diagram illustrating details of the extension adapter of FIG. 1, in accordance with an embodiment of the present invention;

[010] FIG. 5 is a schematic diagram illustrating an operation involving the reading of data in accordance with the extension adapter of FIG. 4;

[011] FIG. 6 is a flow chart illustrating a preferred method of the present invention; and

[012] FIG. 7 is a flow chart further detailing the method of the invention illustrated in FIG. 6.

## DETAILED DESCRIPTION OF THE INVENTION

[013] The present invention provides a method for programming a processor instruction set to include new, extended instructions and for replacing a critical code segment of a computer program with a function that causes the new instruction to execute. As general purpose processors typically do not have programmable instruction sets, the present invention will be described with reference to the programmable processing hardware of FIG. 1, though it will be appreciated that the invention is not so limited and can be used in conjunction with other suitable programmable processing hardware.

[014] FIG. 1 is a schematic drawing of an exemplary programmable processing system 110 including a processor core 120, a programmable logic device (PLD) 130, and an extension adapter 140 that couples the programmable logic device 130 to the processor core 120. The processor core 120 can include optional features such as additional coprocessors, write buffers, exception handling features, debug handling features, read only memory (ROM), etc. The processor core 120 provides standard processing capabilities such as a standard (native) instruction set that provides a set of instructions that the processor core 120 is designed to recognize and execute. Typical instructions include arithmetic functions such as add, subtract, and multiply, as well as load instructions, store instructions, and so forth. These instructions are hard-coded into the silicon and cannot be modified. One example of a suitable processor core 120 is the Xtensa® V (T1050) processor, from Tensilica, Inc., of Santa Clara, California.

[015] Programmable logic device (PLD) 130 includes programmable logic for enabling application-specific instructions (“instruction extensions”) to be stored and executed. Because it is programmable, the instruction set of programmable logic device 130 can be readily configured to include instruction extensions that are tailored to a specific application. In some embodiments

the programmable logic device (PLD) 130 runs at a slower clock speed than processor core 120. In these embodiments the cycle length of the programmable logic device 130 can be a multiple of the clock cycle of the processor core 120.

[016] Extension adapter 140 provides an interface between the programmable logic device 130 and the processor core 120. Extension adapter 140 receives instructions and determines whether the instructions should be directed to the programmable logic device 130 or the processor core 120. In some embodiments extension adapter 140 provides an interface between a plurality of programmable logic devices 130 and processor cores 120. Extension adapter 140 can be implemented, for example, in Application Specific Integrated Circuit (ASIC) logic.

[017] Extension adapter 140 in combination with PLD 130 provide logic that allows users to extend the native instruction set defined by the processor core 120. It is noteworthy that the instruction execution itself is implemented in one or more of programmable logic devices 130. Extension adapter 140 interfaces one or more programmable logic devices 130 to processor core 120 and controls dataflow.

[018] FIG. 2 illustrates one embodiment of a programmable logic device (PLD) 130. As shown, PLD 130 includes a plurality of cluster blocks 202 arranged in rows and columns. Data is communicated between cluster blocks 202 by means of a global interconnect 204. As shown, the global interconnect 204 also communicates data and dynamic configuration information used or output by PLD 130 with other devices including extension adapter 140, which data and dynamic configuration information will be described in more detail below. Although generically shown as permitting any two cluster blocks 202 in PLD 130 to communicate directly with each other via global interconnect 204, such interconnections need not be so limited. For example, cluster

blocks 202 can additionally or alternatively have interconnections such that blocks in adjacent rows and/or columns communicate directly with each other.

[019] Although not necessarily part of PLD 130, and preferably separately provided, also shown is configuration memory 206. Configuration memory 206 stores static configurations for PLD 130. The term “memory” is not intended to be construed as limiting. Rather, configuration memory 206 can have various implementations including CMOS static random access memory (SRAM), fused links and slow speed electrically erasable read only memory (EEPROM).

[020] FIG. 3 illustrates a cluster block arrangement that can be used to implement cluster block 202 in FIG. 2. As shown, it includes a plurality of ALU controller (AC) blocks 302 and function cells 304. The AC blocks 302 provide configuration signals for a respective column 310 of function cells 304. In one example of the invention, cluster block 202 includes four columns of four function cells 304, each column including one AC block 302.

[021] FIG. 3 shows paths for sharing data and dynamic configuration information between vertically or horizontally adjacent function cells 304 within cluster block 202, and with other cluster blocks via global interconnect 204. Also shown are horizontal word lines 308 and vertical word lines 306, by which certain or all of the interior function cells 304 may communicate data with other cluster blocks 202, which word lines partially implement global interconnect 204.

[022] Programmable logic device 130 is described in more detail in U.S. Patent Publication Number US 2001/0049816, which is incorporated herein by reference. A suitable programmable logic device 130 is available from Stretch, Inc., of Mountain View, California.

[023] Referring to FIG. 4, extension adapter 140 is shown in greater detail. In one embodiment, extension adapter 140 comprises load/store module 410 and adapter controller



412. In another embodiment, processor core 120, and not extension adapter 140, comprises load/store module 410.

[024] Load/store module 410 is created via a compiler, such as, for example, the Tensilica Instruction Extension (TIE) compiler, which can be obtained from Tensilica, Inc., of Santa Clara, California. TIE is a language that allows a user to describe the functionality of new extended instructions. A designer uses TIE to create a standard set of functions that extend the normal functionality of processor core 120. The TIE code that a designer writes describes the functionality of a series of resources that aid in the interface between processor core 120 and extension adapter 140. Users can therefore add new instructions pre-silicon. Extension adapter 140 functions such that processor core 120 treats user-defined post-silicon, extended instructions as if they were native instructions to the processor core 120.

[025] Load/store module 410 interfaces with processor core 120 via interface 414. Register file 420 is coupled to interface 414 via processor control and data interface 421 and via PLD control and data interface 423. Adapter controller 412 interfaces with processor core 120 via interface 416. Adapter controller 412 interfaces with PLD 130 via interface 418.

[026] In an exemplary embodiment according to the present invention, load/store module 410 comprises register file 420. Register file 420 is a register file, or collections of registers, that is added by using, for example, the TIE compiler. Register file 420 interfaces with adapter controller 412 via interface 424. In one embodiment, register file 420 is 128 bits wide. In another embodiment, register file 420 is 64 bits wide. However, register file 420 can be of varying widths. It is contemplated that the system can comprise one or more than one register file 420. Adapter controller 412 accesses register file 420. Adapter controller 412 is then used to interface with PLD 130.

**[027]** Load/store module 410 provides fixed instruction functionality. A set of fixed instructions includes instructions for moving data to and from external memory (not shown), into and out of register file 420. This collection of functionality is defined in one embodiment in the TIE language, and is implemented through Tensilica's TIE compiler. It is contemplated that languages other than TIE can be used with the present system. Load/store module 410 contains one or more register files 420 and a set of fixed instructions that give register files 420 access to external memory via load and store instructions. Again, these instructions will be fixed once the silicon is created, and are fully implemented using the standard TIE flow. It is a function of the extension adapter 140 to encapsulate the fixed functionality and manage it with the configurable interface logic.

**[028]** A purpose of load/store module 410 includes declaring the functionality of register file 420, which is basically temporary storage for data that is going to end up being transferred from processor core 120 to PLD 130. Load/store module 410 defines not only register file 420, but also defines how to load and store generic instructions (e.g., Tensilica instructions) of processor core 120 into register file 420. Adapter controller 412 performs the function of interfacing with register file 420. Adapter controller 412 also operates on the data from register file 420 and interfaces register file 420 with PLD 130.

**[029]** In one exemplary methodology, standard load and store instructions are used to move data to and from register file 420. Load instructions issued by the extension adapter 140 retrieve data from memory into register file 420. PLD 130 instructions operate under the control of extension adapter 140 to retrieve stored data from register file 420 to PLD 130 for use in PLD 130 computations or other functional execution. Data resulting from PLD 130 instruction

execution is then returned to register file 420, where store instructions move data from register file 420 to memory via interface 414.

[030] PLD 130 and adapter controller 412 allow a user to add new instructions that change with software on different implementations of the same silicon. For example, a user can add specialized instructions to perform video or audio encoding/decoding. These instructions are not hard-wired into processor core 120, but rather are implemented using the programmably configurable logic of PLD 130. Extension adapter 140 operates as a data and control interface between processor core 120 and PLD 130 by routing extended instructions (i.e., those instructions not part of the original processor core 120 native instruction set) to PLD 130 for execution. Since the logic of PLD 130 is configurable, it is entirely within the scope of the present invention that the configuration of PLD 130 can be changed as frequently as needed to accommodate the inclusion of various extended instructions in application programs being run on the processor core 120.

[031] In one embodiment of the present invention, the inputs and outputs to the extended instruction, as executed in PLD 130, are limited to data transfers between register file 420 or some equivalent special purpose register (processor states) location. In such an embodiment, the number of register file 420 inputs to the PLD 130 computation is limited to a finite number such as three (3), and the number of special purpose register inputs is eight (8) 128-bit registers. The outputs of the PLD 130 computations are directed to register file 420, to equivalent special purpose register, and/or by-passed to processor core 120 for use in execution of the subsequent instruction. In the above embodiment, the number of register file 420 outputs is two (2) and the number of 128-bit, special purpose register outputs is up to eight (8). The extended instruction of the present invention of such an embodiment does not have direct access to data and instruction

memories and caches of the processor core 120. Any data residing in the data and instruction memories or caches of processor core 120 must first be brought into the register file 420 or equivalent special purpose registers using load instructions, before being used by the extended instruction as executed in PLD 130. Such a restriction in the I/O of the extended instruction of this embodiment enables compiler optimization and improved performance. The exact input and output dependencies of the extended instructions are programmed into the C compiler (discussed with reference to FIG. 7) used in scheduling the extended instruction and in allocating the associated register files 420.

[032] It is noteworthy that extension adapter 140 handles the multiplexing of data among register file(s) 420 and PLD 130. Extension adapter 140 manages the timing relationships between register reads and register writes, which are functions of instruction execution length.

[033] It is also noteworthy that the processing system 110 comprises means for ensuring the proper configuration of PLD 130 prior to the execution of a specific extended instruction in the PLD 130. In one example, if the system tries to execute an instruction not included in the instruction set of processor core 120 that has yet to be configured in PLD 130, an exception is generated by the extension adapter 140, resulting in either the proper configuration signals being sent to PLD 130, or in an alternative process, being initiated to deal with the missing configuration.

[034] In keeping with some embodiments according to the present invention, FIG. 5 illustrates an operation involving the reading of data. Configuration memory 510 has a description of what user instructions are adapted to do with respect to the interface to processor core 120. For any instruction that a user creates, those instructions should control processor core 120 in such a way that processor core 120 executes those instructions in similar fashion to native instructions

included in the original processor core 120 instruction set. Configuration memory 510 receives instruction description data 512 (from interface 414 of FIG. 4) as a sequence of binary numbers (e.g., a 24-bit sequence) that is decoded by configuration memory 510 and converted into an address that points to a location in configuration memory 510.

[035] If the instruction description data 512 describes a normal add, subtract, etc. contained in the native instruction set of processor core 120, then configuration memory 510 does not do anything with the instruction. However, if the instruction description data 512 describes an extended instruction that PLD 130 is to execute, then configuration memory 510 returns configuration information 514 back to processor core 120 to indicate this is a valid instruction. Extension adapter 140 will thereafter operate on the extended instruction in cooperation with PLD 130 so that to processor core 120 it appears that the extended instruction is identical in form to a native instruction of processor core 120.

[036] Configuration information 514 is a sequence of data from configuration memory 510, some of which goes to processor core 120 via interface 516. Some of configuration information 514 is transmitted to the ReadAddr 518 (read address) input of register file 420 via interface 424. Data from ReadData 520 (read data) of register file 220 is also carried on interface 424. In this example, configuration information 514 includes the address within register file 420 that an extended instruction needs to be sent to PLD 130 via interface 418.

[037] FIG. 6 is a flow chart illustrating an exemplary embodiment 600 of the method of the invention. The method begins by defining a program in step 610. The program can be defined in a standard programming language that is familiar to computer programmers such as C++.

[038] Thereafter, in step 620, the program is compiled to convert the program from the programming language in which it was written into a machine language that is recognizable by

the processor core 120 (FIG. 1). It will be appreciated that the present method is intended to be iterative, as can be seen from FIG. 6, and that successive iterations initially return to step 620. Whereas in the first pass through step 620 a standard compiler, such as a C++ compiler, compiles the program, in successive iterations an additional extension compiler is also employed, as is discussed elsewhere herein.

[039] Next, in step 630 the compiled program is profiled. Profiling includes executing the compiled program and determining how much time would be expended executing each of the various operations of the program. Profiling in step 630 is preferably performed using a software simulation tool (not shown) that mimics the operation of the processor core 120. Such processor simulators are well known in the art, and each simulator is unique to the processor core 120 being simulated. Alternatively, profiling 630 can occur using a hardware emulator (not shown) or some combination of hardware and software. Hardware emulation is particularly useful in applications where specific timing issues are of concern to the designer.

[040] As in step 620, because the method is iterative, the first pass through step 630 is different than in successive iterations. In the first pass through step 630 the compiled program is executed or simulated solely on the processor core 120 to provide a baseline against which improvements in successive iterations can be measured. It should be noted that some of the more time consuming operations that are typically identified by profiling involve nested loops.

[041] In step 640 a determination is made as to the acceptability of the performance of the program. If the performance is acceptable then the method ends. Otherwise, the method continues to step 650. Generally, in the first pass through step 640 the performance will not be acceptable since no effort has yet been made to optimize the program. In successive iterations, performance can be judged against either subjective or objective standards. In some instances

the program needs to be optimized so that it can return data according to the timing requirements of other programs with which it interfaces. In other instances merely a faster processing speed is desired from the program. In these latter instances, at each iteration the performance is compared to the performance from the prior iteration to determine whether the most recent iteration returned a further improvement. If no further improvement is achieved by a successive iteration, or if the improvement is sufficiently trivial, the performance is deemed to be acceptable and the method ends.

**[042]** In step 650 one or more critical code segments are identified by reviewing the results of the profiling performed in step 630. A critical code segment is a portion of the program's code that took excessive time to execute in step 630. Typically, those code segments that took the longest time to execute are considered to be the most critical and are addressed first by the method. As noted elsewhere, nested loops are frequently identified as critical code segments. If addressing the most critical code segments does not produce acceptable performance in step 640, then in successive iterations the next most critical code segments are identified in step 650.

**[043]** Next, in step 660, the critical code segment identified in step 650 is preferably rewritten as a separate function. An example is illustrative of this process. The following original code segment written in C++ includes a nested loop as the critical code segment:

```
a = 0
for (i = 0; i < 100; i++)
    {for (j = 0; j < 8; j++)
        {a += x[i + j] * y[j];}
    z[i] = a >> k;}
```

The critical code segment can be rewritten as a function, which in the following example is given the name "inner":

```
int inner (short*x, short*y)
```

```

{for (j = 0; j < 8; j ++)
    {a += x[j] * y[j];}
return a >> k;}

```

Advantageously, the function can be written using the same programming language as before. In some embodiments the function does not have to be written from scratch but can instead be selected from a class library (not shown) of pre-defined functions. A class library of pre-defined functions can include functions that might be particularly useful in a certain type of application, such as functions for working with pixel data in video processing applications.

**[044]** In an alternative embodiment, step 660 markers (in C programming, such markers are conventionally referred to as PRAGMAS) are used to demark the beginning and ending of a section of code to be rewritten. Once identified, the demarked section of code is replaced by one, or alternatively, multiple instructions. It should be apparent to those of ordinary skill in the art that the rewriting step of 660 can be performed either manually, or by using an automated conversion tool. Such a conversion tool would be similar to a decompiler; rather than compiling a high level instruction into multiple lower level instructions as in a compiler, the automated conversion tool would convert multiple lower level instructions of the processor core 120 instruction set into one or more complex extended instructions for implementation in PLD 130.

**[045]** Once the critical code segment has been rewritten as a function in step 660, in step 670 the program is revised. The revision includes two operations, designating the function as a code segment to be compiled by an extension compiler and replacing the critical code segment with a statement that calls the function. In some embodiments the function is placed into an extensions file, separate from the program file, that contains the code meant to be compiled by the extension compiler. In other embodiments the function is placed in the program file and demarked in such a way that it can be recognized as intended for the extension compiler so that the standard



compiler will ignore it. Demarking the function in this way can be achieved by a flag before the instruction (e.g., # pragma stretch begin) and a flag after the function (e.g., # pragma stretch end).

[046] As noted, revising the program also includes replacing the critical code segment with a statement that calls the function. Continuing with the prior example, the original code segment that includes the critical code segment can be rewritten by replacing the critical code segment with the statement {z[i] = inner (x + i, y);} as follows:

```
a = 0
for (i = 0; i < 100; i++)
    {z[i] = inner (x + i, y);}
```

Once the program has been revised in step 670 the method returns to step 620 and the program is again compiled. In those embodiments in which the function has been placed in the program file and demarked from the remaining code, a pre-processing tool first finds the function and copies it out to an extensions file.

[047] FIG. 7 illustrates an exemplary sequence of events that occurs during step 620 to compile an extensions file 700 and a program file 710. Initially, the code in the extensions file 700 is compiled by the extension compiler 720. An example of an extension compiler 720 is Stretch C, available from Stretch, Inc. of Mountain View, CA. The extension compiler 720 produces two outputs: a header file 730 and an intermediate file 740 written in a hardware description language such as Verilog HDL. The header file 730 declares a prototype for a specific function used to execute an extended instruction called out by the extension compiler 720 during compilation of the extensions file 700. The header file 730 is a conventional C file that provides instruction information, such as the file name, inputs required, outputs written, and

other required instruction parameters. The intermediate file 740 describes how to implement an instruction in the programmable logic device 130 (FIG. 1) that corresponds to the function. Next, an implementation tool 750 maps the intermediate file 740 to the programmable logic device 130. More specifically, the implementation tool 750 converts the contents of the intermediate file 740 to PLD configuration file 760. Implementation tool 750 generates PLD configuration file 760 consisting of a bit stream that is compiled with program file 710 and header file 730 in standard compiler 770 and incorporated in the executable file 780. This PLD configuration file 760 contains the data that is used by the executable file 780 to configure PLD 130 in much the same way that a Field Programmable Gate Array (FPGA) is programmed.

[048] When the extension adapter 140 encounters a processor core 120 instruction that is not part of the native set, but is rather an extended instruction generated by extension compiler 720, the processor core 120 sends a configuration bit stream to the PLD 130 to appropriately configure the PLD 130 to execute the extended instruction. Thus, the executable file 780 can call the function and the programmable logic device 130 contains an instruction that can perform the function.

[049] Thereafter, in step 630 the program is again profiled. In this and subsequent iterations of the method, in contrast to the first pass through step 630, the extension adapter 140 (FIG. 1) directs the programmable logic device 130 to execute the instruction corresponding to the function when the function is called as the executable file 780 runs. Accordingly, the program executes more efficiently, as will be represented by the profile. Next, in step 640 the performance is again evaluated, and if acceptable the method ends, otherwise it begins a new iteration at step 650.

[050] Returning to step 660, a critical code segment can alternatively be rewritten by selecting a pre-defined function from a class library. The following example is illustrative of pre-defined functions that might be found in a class library according to an embodiment of the present invention, and of an instruction that would be defined from these functions. Typical graphics applications define a pixel by an 8-bit integer for each of three colors such as red, green, and blue. According to the present invention, a class library for graphics applications can include a pre-defined function for red, for example, that defines an unsigned 8-bit declared integer, R, by the function `se_uint<8> R;` and another pre-defined function would define for the pixel an unsigned 24-bit declared integer, P, by the function `se_uint<24> P = (B, G, R);` where B and G correspond to blue and green, respectively. In the C++ programming language integers are generally limited to standard bit lengths such as 8, 16, 32 and 64. Accordingly, the ability to create a 24-bit integer, or any integer with a non-standard number of bits, is a beneficial feature of the present invention. Without the ability to define a pixel as a 24-bit integer, one would have to define the pixel as a 32-bit integer, but at the expense of having to carry 8 unused bits.

[051] The advantage of not having to carry unused bits can be further seen when a number of pixels are assigned to a register with a pre-defined width. For instance, a register, W, that has a 128-bit width can accommodate four 32-bit pixels, but the same register can handle five 24-bit pixels. Expressed as an instruction for a programmable logic device 130, assigning five 24-bit pixels to register W would be expressed as `WR W = (P4, P3, P2, P1, P0).`

[052] In the foregoing specification, the invention is described with reference to specific embodiments thereof, but those skilled in the art will recognize that the invention is not limited thereto. Various features and aspects of the above-described invention may be used individually or jointly. Further, the invention can be utilized in any number of environments and applications

beyond those described herein without departing from the broader spirit and scope of the specification. Accordingly, the specification and drawings are to be regarded as illustrative rather than restrictive.